A Staircase Structure for Scalable and Efficient Synthesis of Memristor-Aided Logic

Alwin Zulehner[1], Kamalika Datta[2], Indranil Sengupta[3], and Robert Wille[1]
[1]Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
[2]National Institute of Technology Meghalaya, India
[3]Indian Institute of Technology, Kharagpur, India
alwin.zulehner@jku.at, kdatta@nitm.ac.in, isg@iitkgp.ac.in, robert.wille@jku.at

ABSTRACT

The identification of the memristor as fourth fundamental circuit element and, eventually, its fabrication in the HP labs provide new capabilities for in-memory computing. While there already exist sophisticated methods for realizing logic gates with memristors, mapping them to crossbar structures (which can easily be fabricated) still constitutes a challenging task. This is particularly the case since several (complementary) design objectives have to be satisfied, e.g. the design method has to be scalable, should yield designs requiring a low number of timesteps and utilized memristors, and a layout should result that is hardly skewed. However, all solutions proposed thus far only focus on one of these objectives and hardly address the other ones. Consequently, rather imperfect solutions are generated by state-of-the-art design methods for memristor-aided logic thus far. In this work, we propose a corresponding automatic design solution which addresses all these design objectives at once. To this end, a staircase structure is utilized which employs an almost square-like layout and remains perfectly scalable while, at the same time, keeps the number of timesteps and utilized memristors close to the minimum. Experimental evaluations confirm that the proposed approach indeed allows to satisfy all design objectives at once.

1 INTRODUCTION

In 1971, Chua envisioned the memristor as fourth fundamental circuits element [4]. With the property that its resistance value changes depending upon the total amount of charge flowing through it—coupled by the fact that the change in resistance is non-volatile—memristors allow for new capabilities in in-memory computing. First physical realizations were fabricated in the HP labs in 2008 [15]. Due to their small size and power consumption, arrays of memristors (also called crossbars) leveraged research on in-memory computing.

In this work, we focus on memristor based design of in-memory logic, where all operations are conducted in memory without intermediate read-outs or reinitialization—thus, avoiding the requirement of a complex controller (cf. [6]). One design style achieving that—the IMPLY logic design style—was proposed in [10], where material implication operations ($A \rightarrow B$) are realized using two memristors and one resistor, by applying different voltages to the inputs of the memristors. Synthesis approaches using this design style have been presented e.g. in [3, 10, 11, 14].

Recently, also the MAGIC design style [8] has been proposed, where each n-input gate is realized using $n + 1$ memristors. Since this design style has superior performance compared to the IMPLY design style (regarding speed and energy; cf. [8]), we choose to follow the MAGIC design style in this work.

However, mapping a netlist of gates following the MAGIC design style to a crossbar structure in a clever and efficient fashion is a non-trivial task, since all memristors involved in an operation must be located in the same row or column of the memristor array. This requires to include several copy operations in order to move logic values to other memristors, which increase the number of required timesteps as well as the number of utilized memristors—a number which obviously should be kept as small as possible. Besides that, also the resulting layout matters, since the memristors active in a gate operation shall be kept close to each other in order to avoid high resistances by long wires that may lead to incorrect computations. Consequently, design methods for memristors have to tackle several (complementary) design objectives. Unfortunately, previously proposed solutions such as proposed in [5, 6, 17] only focus on one of these design objectives and hardly address the other ones—yielding rather imperfect solutions which are generated by state-of-the-art methods thus far (all that is discussed and illustrated in more detail later in Section 2.3).

In order to overcome this issue, we discuss the existing design objectives as well as the current state of the art in detail (focusing on strengths as well as on weaknesses). Based on that, we propose a mapping approach that utilizes a staircase structure for mapping logic gates to an array of memristors. This method yields mappings that employ an almost perfect square-like layout, while remaining perfectly scalable and keeping the number of timesteps and memristors low. Experimental evaluation confirms that the proposed approach indeed satisfies all design objective at once.

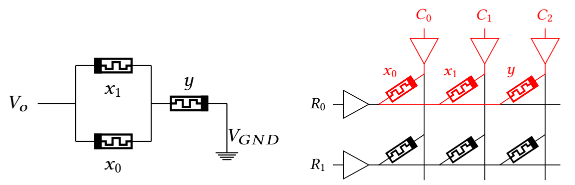
This paper is structured as follows. In Section 2, we review the background of realizing logic using the MAGIC design style as well as current state-of-the-art approaches for realizing these gates in a crossbar structure. In Section 3, we present the proposed mapping approach, followed by a discussion of post-mapping optimizations in Section 4. Section 5 summarizes the results obtained by the proposed approach and compares them to previously propose methods, while Section 6 concludes the paper.

2 BACKGROUND

In order to keep this work self-contained, this section reviews the MAGIC design style, as well as corresponding design objectives including a review on whether and how related works satisfy them thus far. Motivated by the shortcomings of related works, the proposed mapping approach is provided afterwards in Section 3.

2.1 Memristors in the MAGIC Approach

As mentioned earlier, the resistance value of a memristor changes in response to the voltage applied across its terminals. Due to the unique properties of oxygen vacancies that act as charge carriers,



(a) **As standalone logic** (b) **In the memristor crossbar**

Figure 1: MAGIC realization of a NOR gate

the device exhibits a non-volatile behavior and retains its resistance value even when the voltage is withdrawn. The voltage-current characteristic curve of a memristor shows two distinct resistive regions, where one corresponds to low resistance (R_{on}), while the other corresponds to high resistance (R_{off}). The resistance can be switched by applying a suitable voltage across the terminals.

Within the MAGIC design style [8], gate inputs are applied as initial resistive states of the input memristors, while the output memristor is initialized to 0 or 1 (depending on the desired logic gate). During gate evaluation, the output terminal is grounded while a voltage V_o is applied to the input terminals—changing the resistance of the output memristor to the desired value. The magnitude of V_o is determined from the following inequalities

$$\frac{V_{off}}{R_{on}}\left[R_{on} + \frac{R_{off}}{n-1}\,||R_{on}\right] < V_o \tag{1}$$

$$V_o < min\left[V_{off}\cdot\left(1+\frac{R_{off}}{nR_{on}}\right), \left(1+\frac{nR_{on}}{R_{off}}\right)|V_{on}|\right], \tag{2}$$

where V_{off} and V_{on} respectively denote the threshold voltages for switching a memristor to the OFF (i.e. logic 0) and ON (i.e. logic 1) states. By this, e.g. an n-input NOR gate can be realized with $n + 1$ memristors as illustrated by the following example.

Example 1. *A 2-input NOR gate can be realized using memristors as shown in Fig. 1a. The memristors labeled x_0 and x_1 serve as inputs of the gate, (assuming they are initialized accordingly), while the one labeled y holds the output. Before the computation, y has to be initialized with logic 1 (by applying a voltage that sets its resistance to R_{on}). Then, after applying voltage V_o to x_0 and x_1, as well as applying V_{GND} to y as shown in Fig. 1a, the logic value of y changes to zero in case that x_0 or x_1 holds logic 1. Thus, $y = NOR(x_0, x_1)$ afterwards.*

2.2 Logic Design for Memristor Crossbar

To enable in-memory computing, memristors are fabricated in a crossbar array, where the memristors of a row (column) are connected by wires. Each memristor at the junction of a row and a column connects the respective wires (as shown in Fig. 1b). Using MAGIC, only NOR gates can be realized directly in the memristor crossbar [17].[1] Again, an example illustrates the idea.

Example 2. *Fig. 1b shows how NOR gates can be mapped to memristor crossbars. We use the memristor at (R_0, C_2) to compute $y = NOR(x_0, x_1)$.[2] By applying voltage V_o to columns C_0 and C_1 as well as voltage V_{GND} to column C_2, a circuit equivalent to the one shown in Fig. 1a results. Again, we assume that the memristors $M_{0,0}$ and $M_{0,1}$ hold the value of x_0 and x_1, respectively, as well as that $M_{0,2}$ was initialized with logic 1.*

As can be seen in Fig. 1b (highlighted in red), NOR gates can be applied within a *row* of a crossbar Besides that, it often is beneficial to transpose this structure so that NOR gates are applied within a *column*—resulting in a so-called *transposed crossbar* [17]. Then, the voltages that have to be applied in order to conduct the desired operation need to be swapped, i.e. the voltage V_o is applied to the row in which the output is located, while the voltage V_{GND} is applied to the rows that hold the input memristors.

Furthermore, by applying voltages to a column (row) of the crossbar, multiple NOR gate operations are carried out in *all* available rows (columns) of the crossbar structure. While this allows for a parallel execution of gates, it might not always be desired (e.g. if memristor $M_{1,2}$ holds a value which must not be overwritten). Hence, in order to avoid that, one can apply a so-called *isolation voltage* V_{iso} to those rows (columns) in the crossbar that shall be excluded from gate evaluation (e.g. applying V_{iso} to R_1 in Fig. 1b would avoid the execution of a NOR operation in this row). It may be noted that, when multiple gates are mapped to a single row (column), it is not possible to carry out parallel rowwise (columnwise) gate operations.

Following these basic rules, arbitrary functions can be mapped to a memristor crossbar array. First, a representation in terms of NOR gates is constructed. This is accomplished by representing the function to be realized in terms of an *AND-Inverter-Graph* (AIG [7]). After optimizing the AIG (e.g. by using graph rewriting techniques; cf. [13]), the resulting AIG is transformed to a netlist composed of NOR gates and inverters (e.g. using DeMorgan's rules: $a \cdot b = \overline{\overline{a} + \overline{b}}$). More precisely, each node (representing an AND operation) translates directly to a NOR gate by inverting the inputs. After this translation, a dedicated mapping algorithm maps the resulting NOR gates into the memristor crossbar structure. While doing that, it has to be guaranteed that, for each gate, the inputs as well as the output are located in the same row or column. If this is not the case, so called copy operations (composed of two NOT gates) are required to arrange the inputs accordingly. Finally, the result of the mapping algorithm is an assignment of all inputs, intermediate values and outputs (since we use MAGIC gates) to memristors as well as a sequence of operations (i.e. voltages that have to be applied to the rows and columns) required to realize the desired Boolean function.

Note that the effect of graph rewriting techniques, e.g. for minimizing the number of nodes or the depth of the AIG, has intensely been studied for conventional design (c.f. [13]) and can also directly be applied here. Therefore, we do not focus on different optimizations of the function description (the AIG). Instead, we focus on mapping a fixed (and already optimized) netlist composed of NOR gates and inverters to a memristor crossbar. As further benefit, this allows to conduct a fair comparison to previously proposed mapping approaches (cf. Section 5).

2.3 Design Objectives and Related Work

When conducting the mapping from a NOR netlist to a memristor crossbar, the following objectives should be considered:

- *Scalability:* The mapping approach shall be scalable and allow large Boolean functions to be mapped to memristor crossbars.

[1]Since NOT gates are a special case of a NOR gate (one with a single input), they can also be realized directly.
[2]In the following, we denote a memristor at the junction of row R_x and column C_y as $M_{x,y}$.

- *Required Timesteps:* The mapping approach shall aim for a small number of required timesteps. This number has a lower bound defined by the number of stages (i.e. the depth) of the netlist to be mapped. In the ideal case, the outputs of all gates in a stage can be computed in parallel. However, this will hardly be possible since the inputs and the output of the gates have to be arranged in the same row or column.
- *Utilized Memristors:* The mapping approach shall reduce the number of utilized memristors. The fewer memristors used for intermediate computations, the less energy will be consumed. Usually, this metric is correlated to the number of required timesteps, since additional memristors are caused by the requirement of copy operations. Within the MAGIC design style, the number of utilized memristors is bound below by the sum of the number of gates and the number of inputs.
- *Resulting Layout:* The memristors occurring in a gate operation should be close to each other in order to reduce the resistance of the wires connecting them (cf. [12, 17]). Consequently, the dimensions of the bounding box that surrounds the utilized memristors shall be kept as small as possible and hardly skewed.

Several related works exist which conduct the mapping of a NOR netlist to a memristor crossbar [5, 6, 17]. But unfortunately, they either ignore (some of) the design objectives reviewed above or address them in an unsatisfactory fashion. In the following, we briefly discuss the state-of-the-art (assuming that a netlist with n inputs and k gates has to be mapped).

A naive mapping approach has been presented in [17], where all gates are mapped into a single row of the crossbar. This leads to sequential evaluation of the gates but avoids copy operations since all inputs are inherently located in the same row. The approach is scalable, since its complexity grows linearly with the number of gates in the netlist, i.e. $O(k)$. Moreover, the number of required memristors is the minimum, since only one memristor is required for each gate in the netlist (in addition to the memristors needed for the inputs, i.e. in total $n + k$ memristors). However, the number of required timesteps is usually not optimal since always k timesteps are required and no gates can be executed in parallel. Furthermore, the resulting layout is totally skewed since a layout of $1 \times (n + k)$ results.

Example 3. *Consider the NOR netlist shown at the top of Fig. 2. The naive mapping approach results in the crossbar shown at the bottom of Fig. 2. Assuming that the inputs x_0, x_1, x_2, and x_3 are mapped to the first four memristors of the crossbar, we can compute output g_0 of the NOR gate G_0 by applying V_o in columns C_0 and C_1 as well as V_{GND} in column C_4 (cf. Section 2.2). Analogously, we can compute g_1 afterwards. Eventually, we can compute g_2 in row R_0 since the values of g_0 and g_1 are present in the memristors $M_{0,4}$ and $M_{0,5}$, respectively.*

Another approach, called *compact mapping* (proposed in [5]) tries to overcome the issue of the naive approach regarding the resulting layout by additionally mapping gates to columns—resulting in a mapping that grows more equally in the two dimensions. However, the resulting layouts are still rather skewed and the number of required timesteps and memristors is larger than for the naive approach (even though several gates are applied in parallel). This is caused by the large number of copy operations required for aligning the inputs and outputs of the gates.

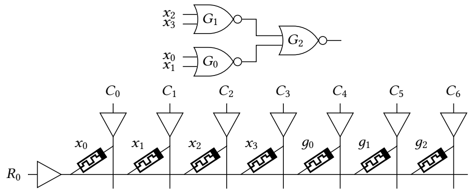


Figure 2: Naive mapping approach

The approach described in [6] instead minimizes the number of required timesteps to reduce the latency of the computation. To this end, they formulate the mapping problem as an optimization problem that is solved with the IBM ILOG CPLEX solver [16]—leading to optimal mappings with respect to the number of required timesteps. However, the solution is not really scalable. In fact, in [6] only benchmarks with up to 21 inputs and at most 10 outputs are listed. Furthermore, no execution times are given in the paper, which makes it hard to estimate the scalability of the approach. With a publicly available implementation of the approach, we could not come up with a solution for functions with fewer than 10 inputs within several days of runtime.

Overall, the current state-of-the-art of mapping a NOR netlist to a memristor crossbar is not capable of satisfying all design objectives at once. In this work, we propose a new scalable mapping approach that keeps the number of required timesteps close to the minimum, uses rather few additional memristors, and yields a layout that is almost a square. How this can be achieved is described in the following section.

3 PROPOSED MAPPING APPROACH

In this section, we propose a dedicated approach for mapping gate netlists to a memristor crossbar.

3.1 General Idea

The general idea is to compute as many gates of the circuit as possible in parallel. All such gates are mapped in different rows, but with the inputs and the output in the same column for each gate. By this, the respective results are located in a single column which then can be used to compute the following gates. This way, each stage of the circuit is alternately mapped in columns and rows—eventually yielding a staircase structure which conforms to the design objectives reviewed above. The general idea is illustrated by means of the following example.

Example 4. *Consider again the NOR netlist shown in Fig. 2. Using the proposed approach results in the mapping shown in Fig. 3. Here, we map the gates g_0 and g_1 to the rows R_0 and R_1 of the crossbar. For both gates (in both rows), we use the columns C_0 and C_1 as inputs, and column C_2 as output. By applying suitable voltages, both gates are computed in parallel.*[3] *Since afterwards the outputs g_0 and g_1 are both available in column C_2, we use this column to compute g_2. We require only two timesteps (instead of three as in naive mapping). Moreover, the bounding box of the mapping is 3×3 instead of 1×7.*

Generalizing this idea leads to a staircase-shaped mapping as shown in Fig. 4. Here, only memristors highlighted in red and blue are used. The computation starts in the bottom right corner of the crossbar. Memristors highlighted blue serve as inputs of gates

[3]Note that we apply V_{iso} in row R_2 to avoid a calculation in this row.

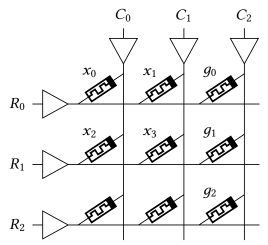


Figure 3: Proposed mapping

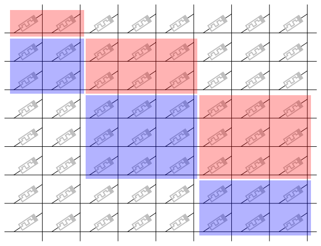


Figure 4: Proposed staircase-structure

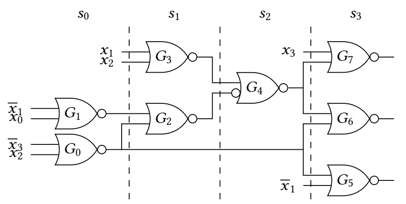


Figure 5: NOR-netlist of a logic function

in odd stages, whereas the outputs of these gates are stored in memristors above the inputs highlighted in red. In contrast, gates of even stages have memristors highlighted in red as inputs as well as the memristors highlighted in blue on the left as outputs. Hence, the outputs of a stage are inherently arranged to serve as inputs for the next stage—resulting in a dedicated mapping algorithm that satisfies the objectives outlined in Section 2.3. More precisely, it is scalable, keeps the number of required timesteps and utilized memristors low, and results in a layout where the bounding box is almost a square.

However, there are still open issues, namely how to map the gates precisely as well as how to handle occurring fanout. This is discussed in detail in the following subsection by means of the netlist shown in Fig. 5. The resulting mapping is shown in Fig. 6, whereas the corresponding operation sequence is shown in Table 1.[4]

3.2 Mapping the Gates of a Stage

As outlined above, we sequentially map the stages of the netlist (in the following denoted by s_i) to the memristor crossbar—starting at the outputs since this allows to handle fanout more efficiently. In fact, whenever we map a gate, we already know all the positions where the fanout must be copied to. If a signal at stage s_j ($j < i$) serves as input for multiple gates of stage s_i, we map all its occurrences into the same row (column) to keep the overhead of handling the fanout as small as possible.

Example 5. *Consider the netlist shown in Fig. 5. First we map stage s_3 which contains the gates G_7, G_6, and G_5. We map these gates to the columns C_0, C_1, and C_2, respectively. To allow for a parallel execution of the three gates, we have to map the output as well as the inputs to the same row for each gate. Fig 6 shows one possible mapping for the gates of stage s_3. The output of the gates (i.e. g_7, g_6, and g_5) are mapped to row R_0 while the inputs are mapped to rows R_1 and R_2, respectively. Since the signals g_4 (i.e. the output of gate g_4) as well as g_0 serve as input for two gates, we map all their occurrences to the same row (i.e. R_1 for g_4 as well as R_2 for g_0).*

After mapping the gates of stage s_3, we continue by mapping s_2 that contains a single gate (G_4) only. We map this gate in row R_1. As input for the gate serve the signals g_3 as well as $\overline{g}_2$ (mapped in the columns C_3 and C_4, respectively). To compute both occurrences of g_4 in row R_1 concurrently, we apply the voltage V_{GND} in columns C_0 and C_1. This way, the fanout of gate G_4 is handled without the need of a copy operation.

Since fanouts across multiple stages are undesired, we do not treat inverters as gates when decomposing the netlist into stages. Instead, we use negated inputs for gates (as e.g. shown for gate G_4 in Fig. 5), i.e. we invert the value stored in one memristor whenever necessary and store the result in an auxiliary memristor. Note that this never causes problems with misaligned inputs, since an inverter has only one input. Our internal parameter study has shown that using negated inputs significantly reduces the number of required timesteps.

Example 5 (continued). *The stage s_1 contains two gates (G_3 and G_2). We map these two gates in columns C_3 and C_4, respectively. For both gates, the inputs are located in rows R_3 and R_4. However, since the output of G_2 is only needed in negated form (i.e. $\overline{g}_2$), an auxiliary memristor is required. We place this memristor in an auxiliary row R_x (indicated in orange in Fig. 6). When determining $\overline{g}_2$, we first compute g_2 using the auxiliary memristor, and invert this value afterwards. The corresponding operation sequence is shown in Table 1.*

3.3 Handling Fanout Across Multiple Stages

As shown in Example 5, fanout of signals from stage s_{i-1} to stage s_i can be handled efficiently by storing the output of a gate in several memristors concurrently (by applying the required ground voltage to the respective columns or rows).[5] However, fanouts across multiple stage of the netlist have to be handled by using auxiliary memristors. We again use them to copy the output of a gate to the location where it is required afterwards.

Example 5 (continued). *After mapping gate G_0 of stage s_0, we have to handle the fanout for this gate. The value g_0 is required at the memristors $M_{2,1}$ and $M_{2,2}$ (highlighted red in Fig. 6). To this end, a copy operation is required. We use an auxiliary memristor $M_{2,4}$ (highlighted in orange in Fig. 6) where we store $\overline{g}_0$ by using an inverter, i.e. $M_{2,4} = INV(M_{4,4})$. By using another inverter we can store g_0 at the desired positions concurrently, by applying voltage V_{GND} to C_1 as well as to C_2. The corresponding operation sequence is shown in Table 1.*

Note that this is an ideal case where the memristor $M_{2,4}$ was not used for storing an intermediate value. If we assume that this memristor cannot be used, more inverters and auxiliary memristors are required. In fact, another option is to use an inverter to compute $M_{0,4} = INV(M_{4,4})$, followed by an inverter that computes $M_{0,3} = M_{0,4}$ ($M_{0,3}$ now holds g_0). Finally, another inverter is needed to compute $M_{2,3} = INV(M_{0,3})$, which enables to set the memristors $M_{2,1}$ and $M_{2,2}$ to g_0 as discussed above.

4 POST-MAPPING OPTIMIZATIONS

The mapping algorithm described in Section 3 yields a mapping of the inputs and gate outputs to the memristors in the crossbar as well

[4]We only list an abstract description of the operations (in terms of NOR gates and inverters) in order to improve readability.

[5]As discussed in [17], this changes the equivalent resistance at the output. Hence, V_o has to be adjusted in Eq. (1) and (2). Some memristor models listed in [9] support gates with a fanout larger than 20—more than the maximum observed in our experiments.

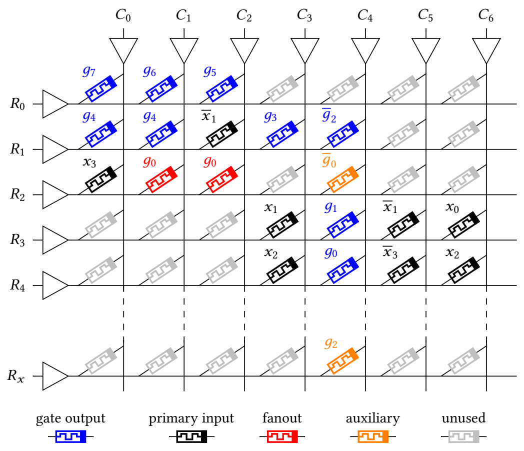


Figure 6: Resulting Mapping

Table 1: Operation Sequence

	t	operations	comment
s_0	1	$M_{3,4} = NOR(M_{3,5}, M_{3,6})$ $M_{4,4} = NOR(M_{4,5}, M_{4,6})$	g_0, g_1
	2	$M_{2,4} = INV(M_{4,4})$	fanout of g_0
	3	$M_{2,1} = M_{2,2} = INV(M_{2,4})$	
s_1	4	$M_{1,3} = NOR(M_{3,3}, M_{4,3})$	g_3
	5	$M_{x,4} = NOR(M_{3,4}, M_{4,4})$	$\overline{g}_2$
	6	$M_{1,4} = INV(M_{x,4})$	
s_2	7	$M_{1,0} = M_{1,1} = NOR(M_{1,3}, M_{1,4})$	g_4
s_3	8	$M_{0,0} = NOR(M_{1,0}, M_{2,0})$ $M_{0,1} = NOR(M_{1,1}, M_{2,1})$ $M_{0,2} = NOR(M_{1,2}, M_{2,2})$	g_5, g_6, g_7

as an operation sequence. In this section, we discuss post-mapping optimizations. While optimizing the mapping to the memristors results in a more compact layout, optimizing the operation sequence reduces the number of required timesteps.

4.1 Optimizing the Mapping

Recall that the mapping algorithm presented in Section 3 yields a staircase-shaped mapping and also uses some auxiliary memristors outside this shape. When we consider the bounding box of this layout in both dimensions, we observe that many memristors that are not utilized.

Example 6. *The bounding box of the mapping shown in Fig. 6 is a rectangle with dimension* 6×7. *However, out of these 42 memristors, only 21 (i.e. the half) are actually utilized.*

In order to increase the density of the utilized area, we try to merge rows as well as columns whenever possible. Merging two rows R_i and R_j (columns C_i and C_j) is possible if in none of the columns (rows) the memristor is used in both rows, i.e if $\forall_k unused(M_{i,k}) \vee unused(M_{j,k})$. Then, we just move the mappings from row R_j (column C_j) to R_i (C_i). Accordingly, we have to adapt the operation sequence.

Example 6 (continued). *In the mapping shown in Fig. 6, we can merge rows* R_0 *and* R_x*, since in* R_x *the memristor* $M_{x,4}$ *is used whereas the memristor* $M_{0,4}$ *is not. Similarly, we can merge column* C_5 *and* C_6 *with columns* C_0 *and* C_1*, respectively. Afterwards, a mapping results where the bounding box is only* 5×5. *Only four out of these 25 memristors are not utilized.*

4.2 Reducing the Number of Timesteps

Besides a more compact mapping, we can also reduce the number of required timesteps in certain cases by inspecting the operation sequence generated by the mapping algorithm. In fact, two operations in the same row (column) that only differ in the target column (row) but have the same source columns (rows) can be merged—resulting in a gate with multiple outputs (similar to the handling of fanouts as described in Section 3.2). Note that this always works since intermediate values stored in the memristors are not overwritten once they are set.

Example 7. *Assume that the mapping algorithm generates a sequence of operations including* $o_i : M_{0,2} = NOR(M_{0,0}, M_{0,1})$ *and* $o_j : M_{0,4} = NOR(M_{0,0}, M_{0,1})$ *with* $i < j$*. Then, we can merge these two operations by replacing* o_i *with* $o'_i : M_{0,2} = M_{0,4} = NOR(M_{0,0}, M_{0,1})$ *and by removing* o_j*.*

Furthermore, there may exist two operations o_i and o_j ($i < j$) which operate on different rows (columns) but use the same columns (rows) as sources and targets. Then, we can again merge these operations to execute both of them in parallel. However, note that merging is not always possible, because the inputs that are required for o_j must already be computed at timestep i. If this is not the case, the two operations cannot be merged.

Example 8. *Assume the mapping algorithm generates a sequence of operations including* $o_{14} : M_{0,2} = NOR(M_{0,0}, M_{0,1})$ *and* $o_{26} : M_{1,2} = NOR(M_{1,0}, M_{1,1})$*. The two operations can only be merged if the memristors* $M_{1,0}$ *and* $M_{1,1}$ *are set to their desired value before timestep 14.*

5 EVALUATION

In this section, we evaluate how well the proposed mapping approach satisfies the design objectives outlined in Section 2.3. To allow for a fair comparison to previous mapping approaches (e.g. proposed in [5, 6, 17]), we decouple the mapping process from the overall design flow, which additionally includes optimization on the netlist level. To this end, we used ABC [1] to optimize the AIG of the function to be realized, before mapping it to a NOR netlist. Since this NOR netlist serves as input for all mapping approaches we compare to, a fair comparison is ensured. The combinational functions of the ISCAS benchmark suite [2] serve as benchmarks.

The obtained numbers for the design objectives outlined in Section 2.3 are presented in Table 2. For each benchmark, we list the number of primary inputs (*PI*) and primary outputs (*PO*), as well as the number of gates in the optimized netlist (g). Furthermore, we list the number of timesteps t, the number of utilized memristors m, as well as the resulting layout for the naive mapping approach (cf. [17]), the compact mapping approach (cf. [5]), as well as for the approach presented in this paper.[6] Note that we do not list the results of the approach proposed in [6] since this approach is not feasible for benchmarks considered here.[7]

As can be seen in Table 2, the proposed approach clearly requires the fewest timesteps on average. More precisely, we obtain a reduction of the timesteps compared to the naive and the compact

[6]Note that we do not list the runtime since all approaches determine a mapping within a second.

[7]In fact, the authors of [6] confirmed that the scalability of their approach is rather limited and that functions with less than 10 inputs may already require several days of runtime. Despite that, we conducted that evaluations anyway on the benchmarks considered in [6] but, due to their relatively small size, this led to rather meaningless results.

Table 2: Experimental Evaluation

				Naive mapping [17]			Compact mapping [5]			Proposed approach		
name	PI	PO	g	t	m	layout	t	m	layout	t	m	layout
c6288	32	32	2 718	2 718	2 749	2749×1	3 776	3 952	2297×6	3 751	6 369	151×870
c1908	33	25	583	583	615	615×1	1 056	1 150	312×13	517	1 087	83×85
c432	36	7	251	251	286	286×1	349	443	146×9	225	405	22×42
c1355	41	32	606	606	646	646×1	1 072	1 232	359×10	236	1 035	96×63
c499	41	32	606	606	646	646×1	1 155	1 293	323×13	242	1 021	96×44
c3540	50	22	1 422	1 422	1 471	1471×1	2 396	2 554	650×16	1 435	2 764	137×164
c880	60	26	521	521	580	580×1	761	919	383×5	427	889	67×39
c5315	178	123	1 989	1 989	2 166	2166×1	3 295	3 737	1261×11	1 361	3 553	221×136
c7552	207	108	2 345	2 345	2 551	2551×1	3 929	4 514	845×14	2 182	4 461	214×175
c2670	233	140	1 060	1 060	1 292	1292×1	1 490	2 018	664×9	551	1 513	66×92

PI : number of primary inputs PO: number of primary outputs g: number of gates in the optimized netlist t: number of required timesteps
m: number of utilized memristors layout: dimension of the bounding box of utilized memristors

mapping of 26.4% and 54.1% on average, respectively. Considering the number of utilized memristors, the naive mapping approach is the clear winner, which is obvious since it yields mappings where this number is the minimum (using the MAGIC design style). However, the proposed approach is only 64.2% above this minimum and requires even 3.8% fewer memristors than the compact mapping algorithm. Finally, the proposed approach yields layouts that are very close to a square, which is desired to keep the resistance low (cf. Section 2.3). In contrast, the layout resulting from naive mapping is totally skewed, and also the layouts obtained by the compact approach are only marginally better.

Overall the mapping approach presented in this paper clearly outperforms the approaches proposed in [5, 6, 17] when considering *all* design objectives outlined in Section 2.3. For example, the approach proposed in [6] yields circuits where the number of timesteps is the minimum, but is not scalable at all. In contrast, the naive mapping approach is scalable and yields a mapping where the number of utilized memristors is the minimum, but generates a mapping layout that is totally skewed and, hence, cannot be used for real applications. The compact approach proposed in [5] tries to overcome this issue, but does not give compelling results either since the layout is still quite skewed in many cases and the number of required timesteps and utilized memristors is quite high. Hence, the proposed approach is the only one that targets all design objectives outlined in Section 2.3 in an integrated fashion at once—even though it may have slight drawbacks compared to other methods when looking at one certain design objective only.

6 CONCLUSIONS

In this paper, we have proposed an approach that maps logic functions to memristor crossbar following the MAGIC design style. This constitutes a non-trivial task due to the constraint that all memristors that are active when applying a gate operation either have to be located in the same row or the same column. Therefore, several (contradicting) design objectives have to be considered. In fact, a scalable solution is desired, which keeps the number of required timesteps and utilized memristors low, while generating a mapping whose bounding box has small dimensions and is hardly skewed. While previously proposed solutions only focused on one certain of these objectives and hardly addressed the others, the approach proposed in this paper satisfies all design objectives at once by utilizing a staircase structure when mapping NOR netlists to a crossbar. Experimental evaluations confirmed that the proposed approach indeed is the only one that targets all design objective at once.

ACKNOWLEDGEMENT

This work was supported fully by the Dept. of Science and Technology, Govt. of India, for the project "Development of CAD Tools for Synthesis, Optimization and Verification of Digital Circuits using Memristors" (Grant No. INT/AUSTRIA/BMWF/P-02/2017), and by the Austrian Agency for International Cooperation in Education and Research (OeAD, Grant No. IN 08/2017).

REFERENCES

[1] R. K. Brayton and A. Mishchenko. ABC: an academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40, 2010.

[2] F. Brglez and H. Fujiwara. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran. In *Int'l Symp. Circuits and Systems (ISCAS 85)*, pages 677–692. IEEE Press, Piscataway, N.J., 1985.

[3] S. Chakraborti, P. V. Chowdhary, K. Datta, and I. Sengupta. BDD based synthesis of Boolean functions using memristors. In *Int'l Design & Test Symp.*, pages 136–141, Dec 2014.

[4] L. Chua. Memristor – the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, Sep 1971.

[5] R. Gharpinde, P. L. Thangkhiew, K. Datta, and I. Sengupta. A scalable in-memory logic synthesis approach using memristor crossbar. *IEEE Trans. VLSI Syst.*, 26(2):355–366, 2018.

[6] R. B. Hur, N. Wald, N. Talati, and S. Kvatinsky. SIMPLE MAGIC: Synthesis and in-memory mapping of logic execution for memristor-aided logic. In *Int'l Conf. on CAD*, pages 225–232, 2017.

[7] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[8] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. MAGIC – memristor-aided logic. *IEEE Trans. on Circuits and Systems*, 61-II(11):895–899, 2014.

[9] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny. VTEAM: A general model for voltage-controlled memristors. *IEEE Trans. on Circuits and Systems*, 62-II(8):786–790, 2015.

[10] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. Memristor-based material implication (IMPLY) logic: Design principles and methodologies. *IEEE Trans. on VLSI Systems*, 22(10):2054–2066, Oct 2014.

[11] F. Lalchhandama, B. G. Sapui, and K. Datta. An improved approach for the synthesis of boolean functions using memristor based IMPLY and INVERSE-IMPLY gates. In *IEEE Computer Society Annual Symp. on VLSI (ISVLSI)*, pages 319–324, July 2016.

[12] J. Liang, S. Yeh, S. S. Wong, and H.-S. P. Wong. Effect of wordline/bitline scaling on the performance, energy consumption, and reliability of cross-point memory array. *J. Emerg. Technol. Comput. Syst.*, 9(1):9, 2013.

[13] A. Mishchenko, S. Chatterjee, and R. K. Brayton. Dag-aware AIG rewriting a fresh look at combinational logic synthesis. In *Design Automation Conf.*, 2006.

[14] A. Raghuvanshi and M. A. Perkowski. Logic synthesis and a generalized notation for memristor-realized material implication gates. In *Int'l Conf. on CAD*, pages 470–477, 2014.

[15] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *nature*, 453(7191):80, 2008.

[16] I. I. C. O. Studio. Using the CPLEXR callable library and CPLEX barrier and mixed integer solver options, 2011.

[17] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky. Logic design within memristive memories using memristor-aided logic (magic). *IEEE Trans. on Nanotechnology*, 15(4):635–650, 2016.